Introduction aux AVR

Julien Le Guen

6 février 2007

Résumé

Ce document est une introduction à la programmation des AVR, notamment la famille ATmega. Il présente brièvement ces microcontrôleurs, puis explique en détail leur utilisation avec le matériel disponible au club Esial RobotiK. Ce document s'adresse aux électroniciens et informaticiens du club qui ne sont pas familiers avec ce type de microcontrôleur, et deviendra au fur et à mesure de l'année un aide mémoire et un guide de la bonne utilisation des AVR. Chacun est convié à contribuer à ce document pour l'enrichir de ses expériences.

Table des matières

1 Introduction

1.1 Présentation des AVR

Les AVR sont des microcontrôleurs (µC) 8bits RISC d'ATMEL très performants (architecture de Harvard), dont la famille est très étendue. On y trouve des petits (ATtiny11, 8 pattes) et des très gros (ATmega128, 64 pattes, insoudable sans aspirine). Ils ont tous en commun le coeur (CPU), seuls les périphériques, la mémoire flash, la ram, etc... diffèrent. Et c'est là un atout de taille, car un code écrit pour un ATmega8 fonctionnera sans modification majeure sur un ATmega32 par exemple (à une recompilation près). Un autre avantage en leur faveur face à d'autres µC comme les PICs par exemple, c'est la suite de logiciel libres permettant de compiler, programmer, débugger les AVR, en live, in situ. Nous reviendrons sur cette notion plus loin.

1.2 Environnement de développement

Vous vous y attendez, on ne va pas programmer en assembleur (quoique ? :). Une suite complète de logiciels libres est disponible pour les AVR, sous linux comme sous windows. On y trouve :

- gcc (`gcc-avr`)
- la libc kivabien (`avr-libc`)
- les binutils adaptés (`binutils-avr`)
- en option, mais on va s'en servir, on peut utiliser diverses bibliothèques (Aversive dans notre cas)

Pour ceux qui ne sont pas familiers de ces outils, cela sert juste à compiler vos programmes C pour les AVR et de générer un fichier binaire (les .hex) utilisable par le µC. Une fois le fichier source compilé et le binaire créé, il faut le transférer dans la flash du µC (la mémoire qui contient le programme exécutable). Plusieurs possibilités s'offrent à nous, nous n'utiliserons que les deux premières cette année :

- programmation *in-situ* (ISP), c'est à dire qu'on reprogramme le µC alors qu'il est dans le montage final, avec un cable adapté branché que le port parallèle de l'ordinateur
- utilisation du cable JTAG, qui permet de reprogrammer, débugger, exécuter pas à pas, etc...
- utilisation d'un bootloader, petit morceau de code qui reste en mémoire et qui papote via une interface série avec l'ordinateur lorsque l'on veut reprogrammer le µC

2 Premiers pas

Nous allons voir dans cette section comment écrire, compiler et flasher un programme simple pour ATmega32. Tout ce qui suit est cependant valable pour la famille ATmega. Tout d'abord, voyons un programme simple (simpliste même) qui fait clignoter une LED.

Ce programme initialise la patte 5 du port C en sortie, puis boucle indéfiniment autour du code faisant clignoter la led. Bien que très simple il nécessite quelques explications. L'instruction `_BV(nb_bits)` est une macro qui place à 1 le bit `nb_bits`. les opérateurs `|`, `&` et `~` sont, dans l'ordre, le OU bits à bits, le ET bits à bits et la négation.

Listing 1 – led0.c

```
#include <avr/io.h>

void mondelai(unsigned short i)
{
    /* Attente active crade */
    while(i-- >0);
}

int main(void)
{
    /* On met la patte PC5 en sortie */
    DDRC |= _BV(PC5);

    while(1)
    {
        /* Fait clignoter la led */
        PORTC &= ~_BV(PC5);
        mondelai(65000);
        PORTC |= _BV(PC5);
        mondelai(5000);
    }
}
```

Il ne reste plus qu'à compiler et flasher le programme dans la mémoire du µC. Pour ce faire, rien de plus simple il suffit d'utiliser le Makefile avec la commande make. Le code du Makefile est disponible ci-dessous pour information :

Listing 2 – Makefile

```
CC=avr-gcc
FILE=led0
OBJCOPY=avr-objcopy
CFLAGS=-g -mmcu=atmega32 -Wall -Wstrict-prototypes -Os -mcall-
    prologues

# ici c'est pour compiler
all: $(FILE).hex
$(FILE).hex: $(FILE).out
```

```
	$(OBJCOPY) -R .eeprom -O ihex $(FILE).out $(FILE).hex

$(FILE).out: $(FILE).o
	$(CC) $(CFLAGS) -o $(FILE).out -Wl,-Map,$(FILE).map $(FILE
	    ).o

$(FILE).o: $(FILE).c
	$(CC) $(CFLAGS) -c $(FILE).c

clean:
	rm -f *.o *.out *.map *.hex

# ici c'est pour programmer facilement l'avr
prog_erase:
	uisp --erase -dprog=dapa

prog_burn:
	uisp --upload if=$(FILE).hex -dprog=dapa -dno-poll -v

prog_verif:
	uisp --verify if=$(FILE).hex -dprog=dapa -dno-poll -v

prog: prog_erase prog_burn prog_verif

cprog: $(FILE).hex prog
```

Il n'y a pas à s'inquiéter devant la taille de ce fichier. Il suffit de taper `make cprog` pour compiler le fichier led0.c et le flasher dans le µC.

Cependant il faut encore relier l'AVR à l'ordinateur, via le cable ISP. Le branchement est simple :

- fil jaune/noir (16) : patte RESET
- fil brun (1) : patte SCK
- fil blanc (11) : patte MISO
- fil rouge (2) : patte MOSI
- fil gris/noir (25) : patte GND

Ils sont regroupés sur un connecteur 6 points, qu'il faut brancher sur le µC en face des pattes correspondantes (attention au sens !) et le tour est joué. On utilise l'utilitaire `uisp` pour papoter avec le µC.

3 Aller plus loin

Le programme led0.c fonctionne, mais il consomme tout le temps processeur et on ne peut rien faire d'autre en attendant. Nous allons donc modifier le programme pour qu'il utilise les interruptions.

Listing 3 – led0.c

```c
#include <avr/io.h>
#include <avr/interrupt.h>
#include <avr/signal.h>

#define F_CPU    1000000UL    // 1MHz
#include <avr/delay.h>

SIGNAL(SIG_OUTPUT_COMPARE1A)
{
    /* Code de l'interruption */
    PORTC ^= _BV(PC5);
}

int main(void)
{
    /* On configure la patte en sortie */
    DDRC |= _BV(PC5);
    PORTC &= ~_BV(PC5);

    /* On configure le timer qui
     * lancera les interruptions */
    TCCR1B = _BV(CS10) | _BV(CS11) | _BV(WGM12);
    OCR1A = (F_CPU/256);
    TIMSK = _BV(OCIE1A);

    /* On active les interruptions */
    sei();
    /* On attend betement */
    while(1);
}
```

Je vous renvoie à la doc de l'ATmega32 pour la signification des octets de contrôle que nous avons modifié.

4 HowTo configurer les fusibles

Les AVR ont des registres, accessibles uniquement avec un programmateur (ISP ou JTAG), qui permettent de configurer physiquement l'AVR : utilisation ou non du JTAG, de ISP, fréquence de l'oscillateur interne, etc... Ils sont appelés fusibles, et la connaissance de leur fonctionnement est primordiale : si les fusibles sont mal positionnés, on risque de ne plus pouvoir programmer le µC !

4.1 Programmation des fusibles

Les fusibles sont mofifiables par ISP ou JTAG. En ISP on peut utiliser `avrdude` ou `uisp` :

4.1.1 Avrdude

Lire la configuration des fusibles :

```
$ avrdude -p m32 -c dapa -U hfuse:r:-:i -U lfuse:r:-:i
```

Ceci nécessite quelques explications (en plus du `man avrdude` indispensable) :
`-U hfuse:r:-:b`
On accède au HighByteFuse, en lecture (`r`), sans valeur (`-`), format hex intel (`i`)

`-U lfuse:r:-:h`
On accède au LowByteFuse, en lecture (`r`), sans valeur (`-`)

Programmer la valeur d'un fusible :

```
$ avrdude -p m32 -c dapa -U lfuse:w:0x89:i
```

On place la valeur 0x89 dans le LowByteFuse.

4.1.2 uisp

Lire la configuration des fusibles (résultat en héxadécimal) :

Listing 4 – Lecture des fusibles

```
$ uisp -dprog=dapa --rd-fuses
Atmel AVR ATmega32 is found.

Fuse Low Byte        = 0xe1
Fuse High Byte       = 0x99
Fuse Extended Byte   = 0xff
Calibration Byte     = 0xb9 -- Read Only
Lock Bits        = 0xff
    BLB12 -> 1
    BLB11 -> 1
    BLB02 -> 1
    BLB01 -> 1
```

```
LB2 -> 1
LB1 -> 1
```

Programmer la valeur d'un fusible (en héxa, exemple pour le LowByteFuse) :

```
$ uisp -dprog=dapa --wr_fuse_l=0x89
```

4.2 ATmega32

L'ATmega32 est un gros micro, sur lequel le débuggage grâce au JTAG est possible. Le Datasheet liste les bits des fusibles et leur utilité en page 257 :

High Byte	**Bit n°**	**Description**	**Default Value**
OCDEN	7	Enable OCD	1 (u, OCD disabled)
JTAGEN	6	Enable JTAG	0 (p, JTAG enabled)
SPIEN	5	Enable SPI serial program	0 (p, SPI enabled)
CKOPT	4	Oscillator Option	1 (u)
EESAVE	3	EEPROM preserved	1 (u)
BOOTSZ1	2	Select boot size	0 (p)
BOOTSZ0	1	Select boot size	0 (p)
BOOTRST	0	Boot reset vector	1 (u)

Low	**Bit n°**	**Description**	**Default Value**
BODLEVEL	7	Brown-out detector level	1 (u)
BODEN	6	Brown-out detector enabled	1 (u, BOD disabled)
SUT1	5	Select start-up time	1 (u)
SUT0	4	Select start-up time	0 (p)
CKSEL3	3	Select clock source	0 (p)
CKSEL2	2	Select clock source	0 (p)
CKSEL1	1	Select clock source	0 (p)
CKSEL0	0	Select clock source	1 (u)

Il faut bien entendu lire la doc pour bien comprendre la signification de toute cette configuration. Sachez cependant qu'il faut désactiver le JTAG pour pouvoir utiliser la patte PC5.

4.2.1 Configuration du JTAG

Il faut donc écrire un "1" dans le bit JTAGEN pour le désactiver (les fusibles sont programmés à l'état bas "0", et désactivés à l'état haut "1").

Pour désactiver le fusibles JTAGEN, il faut positionner le bit 6 du HighByteFuse, et laisser inchangés les autre bits. Pour éviter les bétises, on va tout d'abort regarder l'état des fusibles :

Listing 5 – Lecture des fusibles

```
$ uisp -dprog=dapa --rd_fuses
Atmel AVR ATmega32 is found.
Fuse Low Byte      = 0xef
Fuse High Byte     = 0x99
```

Le reste des infos ne nous intéressa pas ici. Le HighByte vaut 0x99, soit 0b10011001. En mettant le bit 6 à 1 nous obtenons 0b11011001, soit 0xd9. On va donc écrire cette valeur dans le High Byte Fuse :

Listing 6 – Ecriture des fusibles

```
$ uisp -dprog=dapa --wr_fuse_h=0xd9
Atmel AVR ATmega32 is found.
Fuse High Byte set to 0xd9
```

Nous pouvons désormais utiliser le port C comme n'importe quel port. Attention cependant, le JTAG étant désactivé il faudra le remettre si on veut débugger le code.

4.2.2 Configuration de l'oscillateur

L'ATmega32 sort d'usine avec comme oscillateur par défaut un RC interne calibré à 1MHz. Bien évidemment, ce n'est pas suffisamment précis pour notre application, nous allons donc brancher un Quartz et ses condensateurs aux pattes XTAL1 et XTAL2. Nous devons également configurer les fusibles pour lui dire d'utiliser le Quartz (bien plus précis). Les configurations possibles se trouvent page 25 et suivantes du Datasheet.

Device Clocking Option	**CKSEL3..0**
External Crystal/Ceramic Resonator	1111 - 1010
External Low-frequency Crystal	1001
External RC Oscillator	1000 - 0101
Calibrated Internal RC Oscillator	0100 - 0001
External Clock	0000

Lorsque l'on utilise un Quartz, la fréquence est limitée à 8MHz avec CKOPT non programmé(=1), et 16MHz lorsqu'il est programmé (=0). Voici la table des fusibles suivant la fréquence :

CKOPT	**CKSEL3..1**	**(MHz)**
1	101	0.4 - 0.9
1	110	0.9 - 3.0
1	111	3.0 - 8.0
0	101, 110, 111	1.0 $\leqslant$

Le bit CKSEL0 sera positionné à 1 (Crystal resonator), et les bits SUT1 et SUT0 à 10 (fast rising power), cf. page 27 du Datasheet

Cas concret : Quartz de 10MHz, quelles valeurs de fusible programmer ?
On doit programmer le bit CKOPT (dans le High Byte Fuse) et positionner CKSEL 3..0 à 1111 (dans le Low Byte Fuse). On va modifier les valeurs actuelles des fusibles :

Listing 7 – Lecture des fusibles

```
$ uisp -dprog=dapa --rd_fuses
Atmel AVR ATmega32 is found.
Fuse Low Byte      = 0xef
Fuse High Byte     = 0x99
```

On positionne le bit CKOPT (bit 4), ce qui donne une valeur de 0x89 pour le High Byte.
Le Low Byte vaudra quant-à lui 0xef.

Listing 8 – Ecriture des fusibles

```
$ uisp -dprog=dapa --wr_fuse_h=0x89
$ uisp -dprog=dapa --wr_fuse_l=0xef
```

4.3 ATtiny2313

L'ATtiny est un petit micro, qui n'a pas d'interface JTAG. Cependant il possède 3 registres de fusibles pour configurer le fonctionnement du processeur.

Fuse High Byte	**Bit No**	**Description**	**Default Value**
DWEN	7	debugWIRE Enable	1 (ud)
EESAVE	6	EEPROM memory is preserved	1 (u)
SPIEN(1)	5	Enable Serial Downloading	0 (p)
WDTON(2)	4	Watchdog Timer always on	1 (u)
BODLEVEL2	3	Brown-out Detector trigger level	1 (u)
BODLEVEL1(4)	2	Brown-out Detector trigger level	1 (u)
BODLEVEL0	1	Brown-out Detector trigger level	1 (u)
RSTDISBL	0	External Reset disable	1 (u)

Fuse Low Byte	**Bit No**	**Description**	**Default Value**
BODLEVEL	7	Brown-out Detector trigger level	1 (u)
BODEN	6	Brown-out Detector enable	1 (u)
SUT1	5	Select start-up time	1 (u)
SUT0	4	Select start-up time	0 (p)
CKSEL3	3	Select Clock source	0 (p)
CKSEL2	2	Select Clock source	0 (p)
CKSEL1	1	Select Clock source	0 (p)
CKSEL0	0	Select Clock source	1 (u)

4.3.1 Configuration de l'oscillateur

De même que pour l'ATmega, le Datasheet présente les tableaux de configuration en page 25 et suivantes. Pour configurer l'ATtiny en mode Oscillateur externe 10MHz, il faut placer 1111 dans CKSEL 3..0, et déprogrammer le bit CKDIV8 (qui active la division par 8 du signal d'horloge)ce qui donne comme valeur pour le Low Byte (d'après les valeurs par défaut) 0xef.

5 Compilation avec Aversive

Aversive est un framework de développement pour AVR. Il comprend une bibliothèque de fonctionnalités, utilise l'environnement de compilation gcc-avr + avr-binutils, et permet de programmer les chips avec avrdude ou avarice. Utilisable sous linux ou windows. Nous ne traiterons ici que le cas sous linux.

5.1 Installation

Préparez un dossier prêt à recevoir votre travail. Téléchargez le snapshot d'aversive [1], et décompressez-le. Dans le dossier `aversive` venant de se créer, on trouve les sources de la bibliothèque, divers programmes de test, un `Makefile` d'exemple, .etc...

5.2 Création d'un projet

Vous pouvez vous placer dans `aversive/projets` ou ailleur sur le système de fichiers, c'est équivalent (il faut juste bien renseigner le `Makefile`). Pour notre cas, plaçons nous dans `aversive/projets` et créons un dossier `test`. Il faut alors copier le fichier `Makefile` :

```
cp $(AVERSIVE_DIR)/mk/Makefile_project.template Makefile
```

Il ne reste plus qu'à créer un `main.c`, inclure les bons headers en fonctions des modules que l'on veut utiliser (cf. plus loin)

5.3 Utilisation d'avrdude

Si vous avez besoin, pour une raison ou pour une autre, d'utiliser avrdude à la main, voici quelques informations :

- Effacer le micro (l'option y sert à incrémenter un compteur d'effacement en EEPROM)
  `avrdude -p m32 -c dapa -y -e)`
- Programmer la zone flash du micro (avec fichier.hex)
  `avrdude -p m32 -c dapa -y -U flash:w:fichier.hex`

5.4 Liaison série

Voici comment mettre en place simplement une liaison série entre le montage et l'ordinateur.
Tout d'abord, créez un projet Aversive (ou modifiez un existant). Dans une console, faites un coup de `make menuconfig` et modifiez les options relatives à la liaison série (menu `Communication Modules`, cochez `Uart` et `Create default uart config`). Un fichier `uart_config.h` est automatiquement créé. C'est dans ce fichier que sont définis les paramètres de la liaison. Par défaut, liaison 38400 bauds, 8 bits, 1 stop, pas de parité. Pensez à configurer le PC en conséquence ! (Ctrl-A Z sous minicom).
Dans votre fichier C, incluez `<uart.h>`. Puis initialisez la liaison :

[1] Wiki Aversive : http://wiki.droids-corp.org/mediawiki/index.php/Aversive
Snapshot Aversive : http://zer0.droids-corp.org/aversive_snapshot.tar.gz

Listing 9 – Initialisation de l'UART

```
#include <avr/io.h>
#include <wait.h>
#include <uart.h>
#include <stdio.h>

int main(void)
{
    struct uart_config u;
    uart_init();
    fdevopen(uart0_dev_send, uart0_dev_recv);
    sei();
    uart0_getconf(&u);

    printf("Utilisation de l'UART ! %d\r\n", 42);
    return 0;
}
```

L'appel à `fdevopen` permet de configurer l'UART en read/write, le reste est assez simple. Voilà, vous pouvez désormais débugger au printf() !

Pensez à mettre un MAX232 entre le PC et le µC, sinon il va encore y avoir des morts... Le montage est fait entre le PC et la carte de Stooo, pour vos propres montages, servez vous de la doc [2] et surtout, ne branchez pas le µC directement au PC ! (les niveaux de tension sont différents, et mortels pour l'AVR).

[2] http://www.maxim-ic.com/getds.cfm?qv_pk=1798&ln=en

6 Codes sources

Cette section regroupe des exemples de programmes.

6.1 Chenillard

Listing 10 – chenillard.c

```c
/* Chenillard a la K2000 !!! */

#include <avr/io.h>
#include <avr/interrupt.h>
#include <avr/signal.h>

#define F_CPU   1000000UL
#include <avr/delay.h>

unsigned short sens;

// Routine d'interruption
SIGNAL(SIG_OUTPUT_COMPARE1A)
{
    if(sens) PORTC = PORTC << 1;
    else PORTC = PORTC >> 1;

    // Changement de sens
    if(PORTC == 1) sens = 1;
    if(PORTC == 0x80) sens = 0;
}

int main(void)
{
    // Tout le port C en sortie
    DDRC = 0xFF;
    // Initialisation du chenillard
    PORTC = 0xFF;
    _delay_ms(65000);
    PORTC = 1;
    sens = 1;

    // Initialisation du timer
    TCCR1B = _BV(CS10) | _BV(CS11) | _BV(WGM12);
    OCR1A = (F_CPU/2048);
    TIMSK = _BV(OCIE1A);

    // Mise en place de l'interruption
    sei();

    /* Ici on peut executer du code pour gerer autre chose
     * le chenillard continuera a la meme vitesse, grace
     * aux interruptions. */
    while(1);
}
```

6.2 Utilisation d'un télémètre IR numérique

```
/*
 * Utilisation d'un telemetre GP2D02
 * Pas d'interruption
 */

#include <avr/io.h>

#define F_CPU   1000000UL
#include <util/delay.h>

#define CLK PD1
#define VOUT    PD0

/* Macros clear_bit et set_bit */
#define cb(port, bit)   ((port) &= ~_BV(bit))
#define sb(port, bit)   ((port) |= _BV(bit))

/* Lecture d'une valeur du telemetre
 * Il faut respecter un timing defini dans la doc du telemetre
 * d'ou la presence de _delay_us() */
unsigned char lecture()
{
    unsigned char temp, i;
    // Activation du telemetre
    cb(PORTD, CLK);

    // On attend (betement) que le telemetre veuille bien
    while(!(PIND & _BV(VOUT)));

    // On envoie la clock et on lit la valeur renvoyee
    for(i=0; i<8; i++)
    {
        sb(PORTD, CLK);
        temp = temp << 1;
        _delay_us(100);
        cb(PORTD, CLK);
        _delay_us(100);
        // On positionne le LSB selon la valeur de VOUT
        if(PIND & _BV(VOUT)) sb(temp, 0);
        else cb(temp, 0);
    }

    // tempo en fin d'interrogation
    sb(PORTD, CLK);
    _delay_ms(2);
    cb(PORTD, CLK);
    return temp;
}
```

```
void main(void)
{
    unsigned char valeur;

    /* Initialisation */
    // PORTC en sortie
    DDRC = 0xFF;
    PORTC = 0xFF;

    // VOUT en entree, CLK en sortie
    cb(DDRD, VOUT);
    sb(DDRD, CLK);

    // Initialisation terminee
    _delay_ms(65000);
    PORTC = 0;

    while(1)
    {
        valeur = lecture();
        /* Mise en forme de la valeur renvoyee
         * Ici on affiche un bargraph motrant la distance
         * Plus c'est proche, plus la valeur est elevee */
        valeur /= (255/9);
        switch(valeur)
        {
            case 1: PORTC = 0x01; break;
            case 2: PORTC = 0x03; break;
            case 3: PORTC = 0x07; break;
            case 4: PORTC = 0x0F; break;
            case 5: PORTC = 0x1F; break;
            case 6: PORTC = 0x3F; break;
            case 7: PORTC = 0x7F; break;
            case 8: PORTC = 0xFF; break;
            case 0: PORTC = 0x0; break;
        }http://wiki.droids-corp.org/mediawiki/index.php/
            Aversive

        /* Il faut attendre 70ms entre chaque lecture
         * On pourrait utiliser une interruption
         * toutes les 70ms pour liberer le cpu et faire autre
         * chose que cette bete attente active */
        _delay_ms(70);
    }

}
```

Un télémètre IR GP2D02 est branché à l'avr, et on veut récupérer l'information de distance. Celle ci est affichée sous forme de bargraph. Cet exemple implique la lecture de la valeur d'une patte du µC. Attention, il faut utiliser `PINx` pour lire, et `PORTx` pour écrire, où x est le nom du port.

Exercice

Utiliser les interruptions pour lancer la lecture de la valeur toutes les 70ms. Pour information, la fonction lecture() est construite d'après le datasheet du sharp : http://www.junun.org/MarkIII/datasheets/GP2D02.pdf.

6.3 Utilisation d'une liaison série

Cet exemple utilise la carte de Stoo (Eirbot 2005) généreusement prêtée pour qu'on puisse jouer en attendant la notre :) Rien de compliqué là dedans, mais sert d'aide mémoire.

```
#include <avr/io.h>
#include <wait.h>
#include <uart.h>

#include<stdio.h>

#define LEDA    6
#define LEDB    7

int main(void)
{
    struct uart_config u;
    int i=0;
    uart_init();

    sbi(DDRC, LEDA);
    sbi(DDRC, LEDB);
    sbi(PORTC, LEDA);

    fdevopen(uart0_dev_send, uart0_dev_recv);
    sei();

    uart0_getconf(&u);

    while(1) {
        printf("Esial RobotiK %d\r\n", i++);
        PORTC ^= 0xC0;
        wait_ms(1000);

        printf("%d %d %d %d %d / %d %ld\r\n",
            u.enabled, u.intr_enabled, u.use_double_speed,
            u.parity, u.stop_bits, u.nbits, u.baudrate);
        PORTC ^= 0xC0;
        wait_ms(1000);
    }

    return 0;
}
```

7 Memento

7.1 Programmation C

1. Ne pas confondre le ET logique (`&&`) avec le ET bit à bit (`&`). Les utilisations sont totalement différentes !
2. De même, ne pas confondre le OU logique (||) et le OU bit à bit (|).
3. Pour écrire sur un port, **écrire** dans le registre **PORTx** (x est le nom du port). En revanche pour **lire**, il faut utiliser le registre **PINx** !

7.2 AvrDude

AvrDude est un outil de programmation permettant d'utiliser le JTAG. Petit manuel d'utilisation :

1. Brancher le JTAG sur la cible (dans le bon sens) alim éteinte
2. Vérifier les branchements et les court-circuits éventuels sur la cible (sisi)
3. Allumer le bazar
4. Les commandes commencent par `avrdude -p m32 -c jtag1 -P /dev/ttyUSB0` (pour un ATmega32)
5. Lire les fusibles :
   `avrdude -p m32 -c jtag1 -P /dev/ttyUSB0 -U hfuse:r:-:i -U lfuse:r:-:i`
6. Programmer : utilisez le framework Aversive (ou débrouillez-vous)

7.3 Liens utiles

- Atmel : www.atmel.com/avr
- Datasheet ATmega32 : http://atmel.com/dyn/resources/prod_documents/doc2503.pdf
- Wikibook : http://en.wikibooks.org/wiki/Atmel_AVR
- Efficient C Coding for AVR : http://atmel.com/dyn/resources/prod_documents/doc1497.pdf
- Application Notes : http://atmel.com/dyn/products/app_notes.asp?family_id=607
- PID sur AVR : http://atmel.com/dyn/resources/prod_documents/doc2558.pdf
- AVRfreaks : http://avrfreaks.net/